

Bachelor's Thesis

Bachelor's degree in Industrial Technology Engineering

Power consumption datalogger based on Python and Raspberry Pi

REPORT

Author: Pol J. Planas Pulido
Director: Manuel Moreno Eguílaz
Co-Director: Álvaro Gómez Pau
Period: Fall 2019



Escola Tècnica Superior
d'Enginyeria Industrial de Barcelona



Review

This document describes the process to develop a datalogger based on a Raspberry Pi 3, a small single-board computer, and two INA219, bidirectional current and power monitor.

On the first hand, in this document the main hardware components are briefly detailed. Besides, it is shown how are related with a simple schematic of connections to an easy reader understanding, as well as the most relevant technical information used in this project.

Moreover, it is explained step by step how to install the operating system for the Raspberry Pi and how to configure the Access Point (AP). To control the sensors for datalogging, the INA219 library is required due to it contains the main methods to manage the sensors, hence they are defined in this document.

On the other hand, it is necessary to implement a general program controller to manage all the datalogger functions and communications. It is shown how this is achieved by creating its own code. This program allows the interaction between the datalogger and the user to send and receive commands such as start/stop or to change the configuration, for example. It is of great importance to note that the language of these code lines is Python 3.

Finally, once the implementation is done, several tests to prove the system works properly are included. One of the main features of this device is that it can save data in a file or different files as well as receive commands from the user, hence it is demonstrated that the device can work in different ways and achieved the proposed objective.

Summary

REVIEW	3
SUMMARY	4
1. GLOSSARY	7
2. PREFACE	8
2.1. Origin of the project.....	8
2.2. Motivation.....	8
2.3. Previous requirements	8
3. INTRODUCTION	9
3.1. Objectives of the project	9
3.2. Scope of the project.....	9
3.3. Contextualization	9
4. PROJECT	10
4.1. Hardware	10
4.1.1. Raspberry Pi 3 Model B	10
4.1.2. Connector pin GPIO.....	11
4.1.3. INA219.....	13
4.1.3.1. Pin configuration and Functions	14
4.1.3.2. Bus Overview	14
4.1.3.3. Specifications	15
4.1.3.4. Recommended operating conditions.....	15
4.1.3.5. Basic ADC Functions	15
4.1.3.6. Power measurement.....	16
4.1.3.7. PGA function	16
4.1.3.8. Programming the calibration register.....	16
4.1.3.9. Simple Current Shunt Monitor Usage (No Programming Necessary)...	17
4.1.3.10. Serial interfaces	17
4.1.4. Schematic of connections	18
4.2. Installation.....	19
4.2.1. Install and set up Raspbian.....	19
4.2.2. Installation of libraries.....	22

4.2.2.1. INA219's library	23
4.3. Access Point.....	25
4.4. How to connect with a PC or another device	28
4.5. Schematic of communication	29
4.6. Default modules	30
4.6.1. Socket module	30
4.6.2. Threading module	31
4.6.3. Time module	32
4.6.4. Datetime module.....	33
4.7. Code	33
4.7.1. Class Datalogger	34
4.7.1.1. __init__.....	34
4.7.1.2. Sockets	35
4.7.1.3. Start/stop.....	38
4.7.1.4. Sample_conf	38
4.7.1.5. Setup_conf	38
4.7.1.6. Channel_conf	38
4.7.1.7. Short_term / Long_term	38
4.7.1.8. Transfer	40
4.7.2. Class Client.....	40
4.7.2.1. __init__.....	40
4.7.2.2. Connection	40
4.7.2.3. TransferClient.....	41
4.7.3. Class threadloopstartstop	41
4.7.4. Interpolation Matlab script	42
5. TEST AND VALIDATION	44
5.1. Short_term test.....	44
5.2. Long_term test	45
5.3. Start / Stop test.....	45
5.4. Sample_conf test.....	46
5.5. Sockets and transfer test.....	47
5.6. Multi-function test	48
6. DIFFICULTIES AND SOLUTIONS	49

7. BUDGET	50
8. ENVIRONMENTAL IMPACT	51
CONCLUSIONS AND FUTURE WORK	52
ACKNOWLEDGEMENT	53
REFERENCES	54

1. Glossary

- AP: access point
- CSI: Camera Serial interface
- DIY: “Do it yourself ”
- DSI: Display Serial Interface
- GPIO: General Purpose Input/Output
- OS: operating system
- PCB: printed circuit boards
- RoHs: Restriction of Hazardous Substances
- SCL: Serial Clock Line
- SDA: Serial Data Address

2. Preface

2.1. Origin of the project

AMBER is a Research Center focused on Innovation, Development and Validation of High Voltage equipment in the field of Energy Transmission systems. AMBER is part of the Polytechnic University of Catalonia (UPC) and is managed from the MCIA research group [1].

This project arises from the need of this research center to obtain real-time data to subsequently calculate the power and efficiency of several Internet of Things boards with solar and thermal harvesting developed in AMBER. This was the opportunity to perform the project.

2.2. Motivation

In the first place, in the course of my education as an engineer, I have realized that basic computing and electronics were the once I enjoyed the most. That is the reason I want to improve and expand my skills in those fields.

Secondly, our world is based on software, programming and electronics, and the future will be completely technological so that is an opportunity to initiate in this field.

Finally, I decided to work on this project because it was a challenge for me to learn how I could use my knowledge in a practical and real exercise, and besides the project encompass all my interests.

2.3. Previous requirements

This project is mainly based on the programming of a Raspberry Pi to work as a datalogger centre and that it can interact and perform several desired features.

On the one hand, to be able to develop this project, prior programming knowledge was necessary, in this case, the python language. The main module where the functions are programmed, the data logger script and the server-client sockets are examples of python programming. In this bachelor's degree, I coursed two subjects of basic python programming.

On the other hand, once the data has been logged, the corresponding data files are sent to the client for further processing. Hence, it is necessary to provide an access point where any external device can connect to collect these data files. Therefore, it was required a basic networking knowledge.

3. Introduction

3.1. Objectives of the project

The main purpose of this project is to develop a low cost datalogger to work with harvesting boards so it can measure several features such as voltages, currents or calculate efficiency. It will be done by using a Raspberry Pi 3 and two INA129 High Side DC Current Sensor Breakouts.

At this point, different objectives derived from the main one, such as the possibility of controlling the device remotely and transfer data-files via its own Wi-Fi. Hence, it is necessary to create an access point (AP).

Once the project is finished, it will be a useful tool not only for the harvesting boards but for any device that requires a power and/or efficiency study.

3.2. Scope of the project

This project is focused on developing a low-cost datalogger device that can save several input data in files and transfer them if required. It can also communicate via Wi-Fi with an external user.

The project is mainly based on software but also in hardware. Hence, a Raspberry Pi 3 is used as a server and data center with two INA219 boards. The software is programmed in Python 3 to manage the device.

3.3. Contextualization

A datalogger is a common device in the industry. Many commercial devices can monitor several signals. The main problem I realise is that these commercial dataloggers are very expensive and not affordable for small businesses or individual users, among others.

This project pretends to develop a datalogger with the main functions of the commercial devices but much more affordable. Hence, the objective is to create a low-cost datalogger.

4. Project

4.1. Hardware

4.1.1. Raspberry Pi 3 Model B

The use of Raspberry Pi is of great interest for the development of this project due to its great versatility and compact design.

The Raspberry Pi [2] is a series of small single-board computers developed in the United Kingdom by the Raspberry Pi Foundation to promote the teaching of basic computer science in schools and developing countries. It does not include peripherals (such as keyboards and mice) or cases.

The Raspberry Pi is, however, an astoundingly versatile device that packs a lot of hardware and software and is perfect for hobby electronics, DIY projects, setting up an inexpensive computer for programming lessons and experiments, and other endeavours.

Raspberry Pi 3 Model B includes [3]:

- 1.2 GHz ARM processor Systems-On-a-Chip (SoC) with integrated 1GB RAM
- 1 HDMI port for digital audio/video output
- 1 3.5mm jack that offers both audio and composite video out
- 4 USB 2.0 ports for connecting input devices and peripheral add-ons
- CSI connector for connecting the camera
- DSI display connector for touch screen
- 1 microSD card reader for loading the operating system
- 1 Ethernet LAN port
- 1 Integrated Wi-Fi/Bluetooth radio antenna
- 1 micro USB power port
- 1 GPIO (General Purpose Input/Output) interface

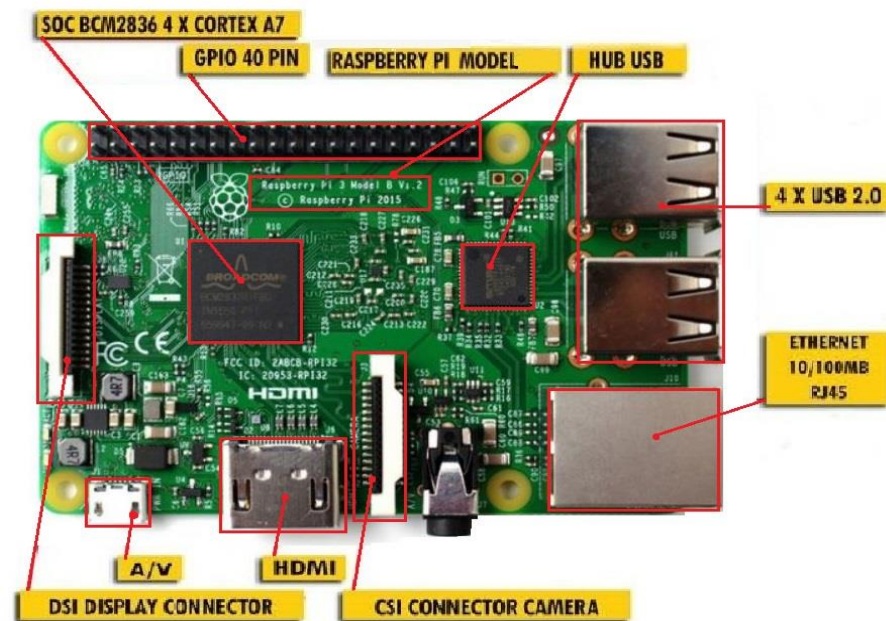


Figure 1. Raspberry Pi Hardware. Source: [3].

4.1.2. Connector pin GPIO

General Purpose Input Output (GPIO) is a general-purpose input and output system. It consists of a series of pins or connections that can be used as inputs or outputs for multiple uses. These pins are included in all Raspberry Pi models although with differences.

It is known that depending on the model of the Raspberry Pi the number of pins could be different, for example, in Raspberry Pi version 1 there are 26 GPIO pins while from Raspberry Pi version 2 and version 3 the number of pins increased to 40, as it is shown in Figure 1. However, compatibility is total, since the first 26 pins retain their original function.

GPIO pins have specific functions, some share functions, and can be grouped as follows (see Figure 2):

- Yellow: 3.3 V source.
- Red: 5 V source.
- Orange: general-purpose inputs/outputs. They can be configured as input or output. High level is 3.3 V, hence 5 V is not tolerated.
- Grey: reserved.
- Black: Ground (GND) connection.
- Blue: transmission through I²C communication protocol (SDA and SCL).
- Green: connected to UART for the conventional serial port.
- Purple: transmission through SPI communication protocol

All pins are unbuffered, hence they do not have buffer protection so the board could be damaged if perform an improper use.

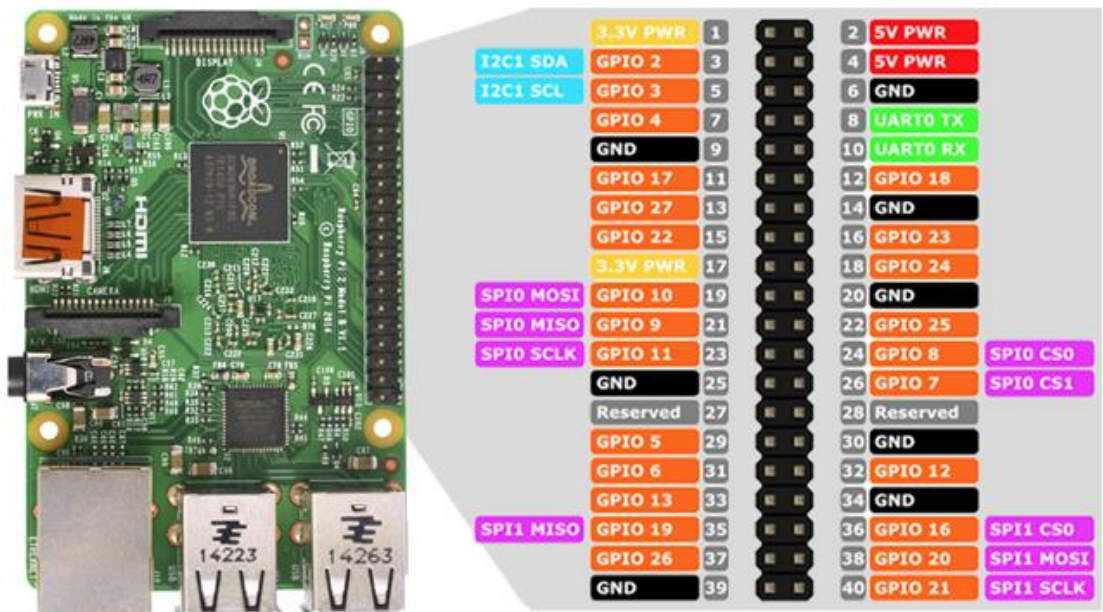


Figure 2. GPIO pins for Raspberry Pi 3. Source: [4].

The pins can be numerated in two different modes: See Figure 3.

- GPIO mode: pins are physically numbered by the place they occupy on the board.
- BCM mode: the pins are numbered by correspondence on the Broadcom chip (CPU of Raspberry Pi).

BOARD	GPIO		GPIO	BOARD
01	3.3v DC Power		DC Power 5v	02
03	GPIO02 (SDA1 , I ² C)		DC Power 5v	04
05	GPIO03 (SCL1 , I ² C)		Ground	06
07	GPIO04 (GPIO_GCLK)		(TXD0) GPIO14	08
09	Ground		(RXD0) GPIO15	10
11	GPIO17 (GPIO_GEN0)		(GPIO_GEN1) GPIO18	12
13	GPIO27 (GPIO_GEN2)		Ground	14
15	GPIO22 (GPIO_GEN3)		(GPIO_GEN4) GPIO23	16
17	3.3v DC Power		(GPIO_GEN5) GPIO24	18
19	GPIO10 (SPI_MOSI)		Ground	20
21	GPIO09 (SPI_MISO)		(GPIO_GEN6) GPIO25	22
23	GPIO11 (SPI_CLK)		(SPI_CE0_N) GPIO08	24
25	Ground		(SPI_CE1_N) GPIO07	26
27	ID_SD (I ² C ID EEPROM)		(I ² C ID EEPROM) ID_SC	28
29	GPIO05		Ground	30
31	GPIO06		GPIO12	32
33	GPIO13		Ground	34
35	GPIO19		GPIO16	36
37	GPIO26		GPIO20	38
39	Ground		GPIO21	40

Figure 3. GPIO pins for Raspberry Pi 3. Source: [4].

4.1.3. INA219

In this section, the technical specifications of the bidirectional current/power monitor INA219 needed for this project are summarized. More information is included in the corresponding datasheet [5]. It is important to emphasize that only the necessary information is shown, not all the datasheet.

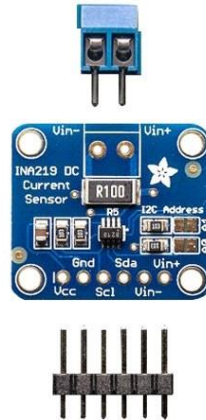


Figure 4. INA219 breakout. Source: [6].

The INA219 is a current shunt and power monitor with an I²C- or SMBUS-compatible interface. It provides digital current, voltage, and power readings necessary for accurate decision-making in precisely-controlled systems. The device monitors both shunt voltage drop and bus supply voltage, with programmable conversion times and filtering. A programmable calibration value, combined with an internal multiplier, enables direct readouts of current in amperes. An additional multiplying register calculates power in watts. The I²C- or SMBUS-compatible interface features 16 programmable addresses. Programmable registers allow flexible configuration for measurement resolution as well as continuous-versus-triggered operation [5].

4.1.3.1. Pin configuration and Functions

The pin configuration is detailed in Table 1. Also, a brief description of every pin is shown.



Pin Functions

PIN			I/O	DESCRIPTION
NAME	SOT-23	SOIC		
IN+	1	8	Analog Input	Positive differential shunt voltage. Connect to positive side of shunt resistor.
IN-	2	7	Analog Input	Negative differential shunt voltage. Connect to negative side of shunt resistor. Bus voltage is measured from this pin to ground.
GND	3	6	Analog	Ground
V _S	4	5	Analog	Power supply, 3 to 5.5 V
SCL	5	4	Digital Input	Serial bus clock line
SDA	6	3	Digital I/O	Serial bus data line
A0	7	2	Digital Input	Address pin.
A1	8	1	Digital Input	Address pin.

Table 1. INA219's pin configuration. Source: [5].

4.1.3.2. Bus Overview

The INA219 offers compatibility with both I²C and SMBus interfaces. The I²C and SMBus protocols are essentially compatible with one another.

Two bidirectional lines, SCL and SDA, connect the INA219 to the bus. Both SCL and SDA are open-drain connections.

The device that initiates the transfer is called a master, and the devices controlled by the master are slaves. The bus must be controlled by a master device that generates the serial clock (SCL), controls the bus access, and generates START and STOP conditions.

To address a specific device, the master initiates a START condition by pulling the data signal line (SDA) from a HIGH to a LOW logic level while SCL is HIGH. All slaves on the bus shift in the slave address byte on the rising edge of SCL, with the last bit indicating whether a read or write operation is intended. During the ninth clock pulse, the slave being addressed responds to the master by generating an Acknowledge and pulling SDA LOW.

Data transfer is then initiated and eight bits of data are sent, followed by an Acknowledge bit. During data transfer, SDA must remain stable while SCL is HIGH. Any change in SDA while SCL is HIGH is interpreted as a START or STOP condition.

Once all data have been transferred, the master generates a STOP condition, indicated by pulling SDA from LOW to HIGH while SCL is HIGH. The INA219 includes a 28-ms timeout on its interface to prevent locking up an SMBus [5].

4.1.3.3. Specifications

Over operating free-air temperature range (unless otherwise noted) [5].

		MIN	MAX	UNIT
V_S	Supply voltage		6	V
Analog Inputs IN+, IN–	Differential ($V_{IN+} - V_{IN-}$) ⁽¹⁾	–26	26	V
	Common-mode ($V_{IN+} + V_{IN-}$) / 2	–0.3	26	V
SDA		GND – 0.3	6	V
SCL		GND – 0.3	$V_S + 0.3$	V
Input current into any pin			5	mA
Open-drain digital output current			10	mA
Operating temperature		–40	125	°C
T_J	Junction temperature		150	°C
T_{stg}	Storage temperature	–65	150	°C

Table 2. Specifications. Source: [5].

(1) V_{IN+} and V_{IN-} may have a differential voltage of –26 to 26 V; however, the voltage at these pins must not exceed the range –0.3 to 26 V.

4.1.3.4. Recommended operating conditions

Over operating free-air temperature range (unless otherwise noted) [5].

	MIN	NOM	MAX	UNIT
V_{CM}		12		V
V_S		3.3		V
T_A	–25		85	°C

Table 3. Recommended operation conditions. Source: [5].

4.1.3.5. Basic ADC Functions

The two analog inputs to the INA219, IN+ and IN–, are connected to a shunt resistor in the bus of interest [5]. The INA219 is typically powered by a separate supply from 3 to 5.5 V. The bus being sensed can vary from 0 to 26 V. There are no special considerations for power-supply sequencing.

The INA219 senses the small drop across the shunt for shunt voltage and senses the voltage concerning the ground from IN– for the bus voltage.

When the INA219 is in the normal operating mode (that is, MODE bits of the Configuration register are set to 111), it continuously converts the shunt voltage up to the number set in the shunt voltage averaging function (Configuration register, SADC bits). The device then converts the bus voltage up to the number set in the bus voltage averaging (Configuration register, BADC bits). The Mode control in the Configuration register also permits selecting modes to convert only voltage or current, either continuously or in response to an event (triggered).

4.1.3.6. Power measurement

Current and bus voltage are converted at different points in time, depending on the resolution and averaging mode settings. For instance, when configured for 12-bit and 128 sample averaging, up to 68 ms in time between sampling these two values is possible. Again, these calculations are performed in the background and do not add to the overall conversion time.

4.1.3.7. PGA function

If larger full-scale shunt voltages are desired, the INA219 provides a PGA function that increases the full-scale range up to 2, 4, or 8 times (320 mV). Additionally, the bus voltage measurement has two full-scale ranges: 16 or 32 V.

4.1.3.8. Programming the calibration register

The Calibration Register is calculated based on (1). This equation includes the term Current_LSB, which is the programmed value for the LSB for the Current Register (04h). This value is used to convert the value in the Current Register (04h) to the actual current in amperes. The highest resolution for the Current Register (04h) can be obtained by using the smallest allowable Current_LSB based on the maximum expected current as shown in (2). While this value yields the highest resolution, it is common to select a value for the Current_LSB to the nearest round number above this value to simplify the conversion of the Current Register (04h) and Power Register (03h) to amperes and watts respectively. The RSHUNT term is the value of the external shunt used to develop the differential voltage across the input pins. The Power Register (03h) is internally set to be 20 times the programmed Current_LSB see (3).

$$Cal = trunc \frac{0.04096}{Current_LSB \times R_{SHUNT}} \quad (1)$$

where 0.04096 is an internal fixed value used to ensure scaling is maintained properly.

$$Current_LSB = \frac{Maximum\ Expected\ Current}{2^{15}} \quad (2)$$

$$Power_LSB = 20 \times Current_LSB \quad (3)$$

Shunt voltage is calculated by multiplying the Shunt Voltage Register contents with the Shunt Voltage LSB of 10 μ V.

After programming the Calibration Register, the value expected in the Current Register (04h) can be calculated by multiplying the Shunt Voltage register contents by the Calibration Register and then dividing by 4096, as shown in (4). To obtain a value in amperes the Current register value is multiplied by the programmed Current_LSB.

$$\text{Current Register} = \frac{\text{Shunt Voltage Register} \times \text{Calibration Register}}{4096} \quad (4)$$

The value expected in the Power register (03h) can be calculated as shown in (5).

$$\text{Power Register} = \frac{\text{Current Register} \times \text{Bus Voltage Register}}{5000} \quad (5)$$

4.1.3.9. Simple Current Shunt Monitor Usage (No Programming Necessary)

The INA219 can be used without any programming if it is only necessary to read a shunt voltage drop and bus voltage with the default 12-bit resolution, 320 mV shunt full-scale range (PGA = /8), 32-V bus full-scale range, and continuous conversion of shunt and bus voltage. Without programming, the current is measured by reading the shunt voltage. The Current register and Power register are only available if the Calibration register contains a programmed value.

4.1.3.10. Serial interfaces

The INA219 operates only as a slave device on the I²C bus and SMBus. Connections to the bus are made through the open-drain I/O lines SDA and SCL. The SDA and SCL pins feature integrated spike suppression filters and Schmitt triggers to minimize the effects of input spikes and bus noise. The INA219 supports the transmission protocol for fast (1- to 400-kHz) and high-speed (1-kHz to 2.56-MHz) modes. All data bytes are transmitted most significant byte first.

4.1.4. Schematic of connections

The main objective of this project is to make a datalogger that stores in a file the necessary measurements for power, efficiency and consumption calculations, among others. For this, two INA219 breakouts are used as explained in previous sections. For good device operation, it is necessary to make a connection circuit as shown in Figure 5.

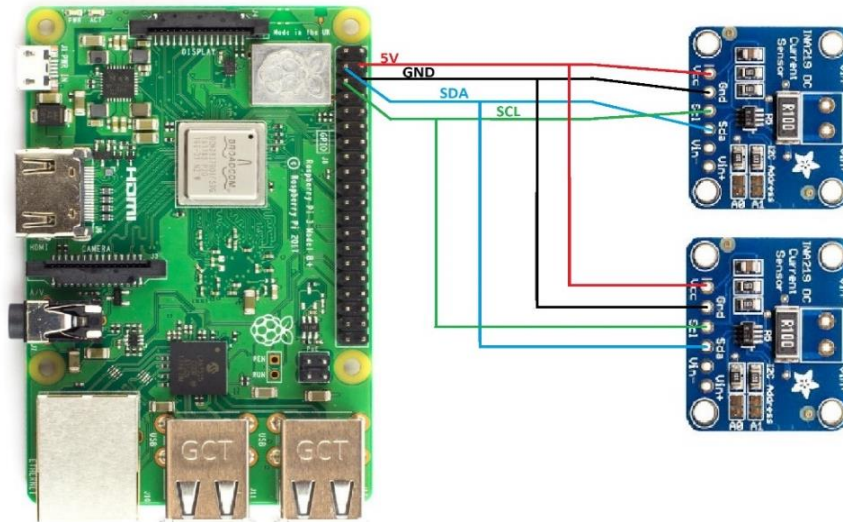


Figure 5. Schematic of connections. Source: own.

The connection between the two INA219 breakouts and the GPIO pins of the Raspberry Pi is made on a PCB as shown in Figure 6, where the tracks are made with soldered tin. Not all tracks are possible to be soldered, hence in the other face of the connection board, some wires are soldered to avoid short circuits.



Figure 6. Connections on the soldered board. Source: own.

4.2. Installation

The development of the configuration for this project is based on existing software that provides an operating system to the Raspberry Pi 3 and libraries used in Python programs that just need to be pre-installed. Raspbian [7] will be used as the default operating system.

4.2.1. Install and set up Raspbian

Raspbian [7] (an ad-hoc Linux distribution) is the recommended operating system for normal use on a Raspberry Pi.

Raspbian is a free operating system based on Debian, optimized for the Raspberry Pi hardware. It comes with over 35,000 packages: precompiled software bundled in an easy-installation format.

For the software installation, a formatted micro SD is required. The micro SD initially has several partitions that need to be removed. Therefore, it is necessary to proceed to execute DiskPart by the command *DiskPart* in the *Run* windows.

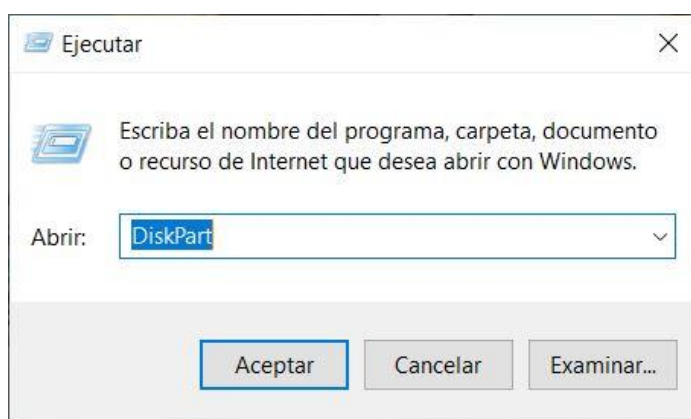


Figure 7. Execute DiskPart command. Source: own.

Following with the commands *list disk* to show all the active disk, then select the disk that needs to be formatted by *sel disk [number of the disk]* and remove all the contents inside by typing *clean*.

Important: Using these instructions will erase everything on the selected drive. If the PC has multiple drives connected, it is also recommended to disconnect them to minimize the chances of selecting the wrong partition.

```
DISKPART> list disk

Núm Disco  Estado      Tamaño  Disp   Din  Gpt
-----
Disco 0    En línea    465 GB   0 B    *
Disco 1    En línea    3900 MB  256 KB

DISKPART> sel disk 1

El disco 1 es ahora el disco seleccionado.

DISKPART> clean

DiskPart ha limpiado el disco satisfactoriamente.

DISKPART>
```

Figure 8. Create a new simple volume for the device. Source: own.

The command *clean* eliminates everything in the drive, hence it needs to implement a new primary partition. Enters the Disk Management window search for the removed device and select *new simple volume* and follow the wizard procedure.

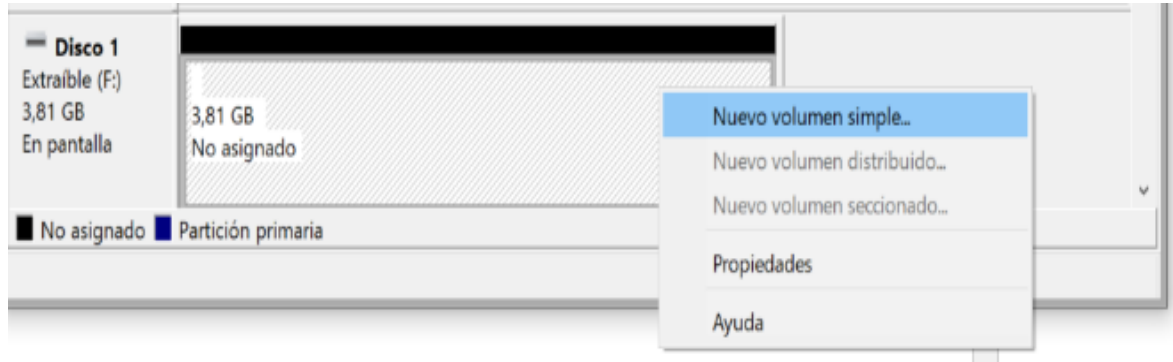


Figure 9. Create a new simple volume for the device. Source: own.

At this point, the micro SD is ready for the software installation. For this process, it is necessary to download the OS image of Raspbian [8]. Proceed to implement the OS image to the micro SD by an external program as BalenaEtcher [9]. It is a very easy-use program with tree steps.

Firstly, selects the OS image from the directory, next selects the device where install Raspbian, and finally select flash.

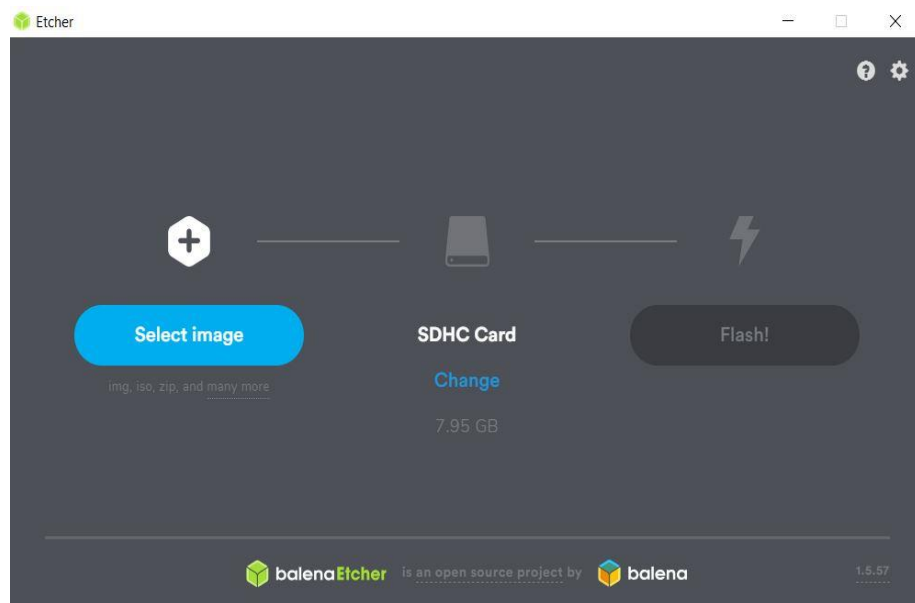


Figure 10. Installing Raspbian with BalenaEtcher. Source: own

Now the Raspbian is installed in the micro SD and ready to run in the Raspberry Pi. To conclude the process, it is necessary to do the basic configuration inserting the micro SD with the OS Raspbian and powering the Raspberry. For an easy configuration connect an HDMI to

a monitor to see the Raspbian desktop. Once the software is running, it displays a window configuration.



Figure 11. Window configuration of Raspbian. Source: own.

It is necessary to follow four steps. In the first instant select the country, language and time zone as shown in Figure 11 and press *Next*. The default user is *pi* and the password is *raspberry*. Hence, proceed to change the password to yield more security.

Finally, the last two steps are to set up a Wi-Fi connection if any network is available and update/upgrade the system. In case of no internet connection it is possible to omit last step and update and upgrade afterwards by the command:

```
sudo apt-get update
sudo apt-get upgrade
```

The installation and configuration of Raspberry Pi is done and prepared to be used.

4.2.2. Installation of libraries

Python library is a collection of functions and methods that allows you to perform many actions without writing your code. Each library in Python contains a huge number of useful modules that you can import for your everyday programming.

Modules written in Python can be classified into three types:

- Created by yourself.
- Created by others or an external source like PyPI [10].
- Preinstalled (already there when you first installed executable).

In the next sections, it is shown the libraries created by external sources used for the project purpose.

4.2.2.1. INA219's library

This Python library supports the INA219 voltage, current and power monitor sensor from Texas Instruments. The library intends to make it easy to use the quite complex functionality of this sensor [11].

The library currently only supports continuous reads of voltage and power, but not triggered reads and supports the detection of overflow in the current/power calculations which results in meaningless values for these readings.

The low power mode of the INA219 is supported, so if only occasional reads are being made in a battery-based system, current consumption can be minimized.

The address of the sensor unless otherwise specified is the default of 0x40. This project uses two sensors. Therefore, the address of the second sensor must be changed to, for example, 0x41 in the code lines.

Note that the bus voltage is that on the load side of the shunt resistor. If the voltage on the supply side is wanted, then it should be added the bus voltage and shunt voltage together, or use the `supply_voltage()` function.

Below, the main functions defined in the INA219 module are shown [11].

- `INA219 ()` constructs the class. The arguments, are:
 - *shunt_ohms*: The value of the shunt resistor in Ohms (mandatory).
 - *max_expected_amps*: The maximum expected current in Amps (optional).
 - *busnum*: The I2C bus number for the device platform, defaults to auto detects 0 or 1 for Raspberry Pi or Beagle bone Black (optional).
 - *address*: The I2C address of the INA219, defaults to 0x40 (optional).
 - *log_level*: Set to logging.INFO to see the detailed calibration calculations and logging.DEBUG to see register operations (optional).
- `configure ()` configures and calibrates how the INA219 will take measurements. The arguments, which are all optional, are:
 - *voltage_range*: The full-scale voltage range, this is either 16 V or 32 V, represented by one of the following constants (optional).
 - `RANGE_16V`: Range zero to 16 Volts

- RANGE_32V: Range zero to 32 Volts (default). Device only supports up to 26V.
- *gain*: The gain, which controls the maximum range of the shunt voltage, represented by one of the following constants (optional).
 - GAIN_1_40MV: Maximum shunt voltage 40 mV
 - GAIN_2_80MV: Maximum shunt voltage 80 mV
 - GAIN_4_160MV: Maximum shunt voltage 160 mV
 - GAIN_8_320MV: Maximum shunt voltage 320 mV
 - GAIN_AUTO: Automatically calculate the gain (default)
- *bus_adc*: The bus ADC resolution (9, 10, 11, or 12-bit), or set the number of samples used when averaging results, represented by one of the following constants (optional).
 - ADC_9BIT: 9 bit, conversion time 84 us.
 - ADC_10BIT: 10 bit, conversion time 148 us.
 - ADC_11BIT: 11 bit, conversion time 276 us.
 - ADC_12BIT: 12 bit, conversion time 532 us (default).
 - ADC_2SAMP: 2 samples at 12 bit, conversion time 1.06 ms.
 - ADC_4SAMP: 4 samples at 12 bit, conversion time 2.13 ms.
 - ADC_8SAMP: 8 samples at 12 bit, conversion time 4.26 ms.
 - ADC_16SAMP: 16 samples at 12 bit, conversion time 8.51 ms
 - ADC_32SAMP: 32 samples at 12 bit, conversion time 17.02 ms.
 - ADC_64SAMP: 64 samples at 12 bit, conversion time 34.05 ms.
 - ADC_128SAMP: 128 samples at 12 bit, conversion time 68.10 ms.
- *shunt_adc*: The shunt ADC resolution (9, 10, 11, or 12-bit), or set the number of samples used when averaging results, represented by one of the following constants (optional).
 - ADC_9BIT: 9 bit, conversion time 84 us.
 - ADC_10BIT: 10 bit, conversion time 148 us.
 - ADC_11BIT: 11 bit, conversion time 276 us.
 - ADC_12BIT: 12 bit, conversion time 532 us (default).
 - ADC_2SAMP: 2 samples at 12 bit, conversion time 1.06 ms.
 - ADC_4SAMP: 4 samples at 12 bit, conversion time 2.13 ms.
 - ADC_8SAMP: 8 samples at 12 bit, conversion time 4.26 ms.
 - ADC_16SAMP: 16 samples at 12 bit, conversion time 8.51 ms
 - ADC_32SAMP: 32 samples at 12 bit, conversion time 17.02 ms.
 - ADC_64SAMP: 64 samples at 12 bit, conversion time 34.05 ms.
 - ADC_128SAMP: 128 samples at 12 bit, conversion time 68.10 ms.

- `voltage ()` Returns the bus voltage in Volts (V).
- `supply_voltage ()` Returns the bus supply voltage in Volts (V). This is the sum of the bus voltage and shunt voltage. A `DeviceRangeError` exception is thrown if current overflow occurs.
- `current ()` Returns the bus current in milliamps (mA). A `DeviceRangeError` exception is thrown if current overflow occurs.
- `power ()` Returns the bus power consumption in milliwatts (mW). A `DeviceRangeError` exception is thrown if current overflow occurs.
- `shunt_voltage ()` Returns the shunt voltage in millivolts (mV). A `DeviceRangeError` exception is thrown if current overflow occurs.
- `current_overflow ()` Returns 'True' if an overflow has occurred. Alternatively handle the `DeviceRangeError` exception as shown in the examples above.
- `sleep ()` Put the INA219 into power down mode.
- `wake ()` Wake the INA219 from power down mode.
- `reset ()` Reset the INA219 to its default configuration.

4.3. Access Point

An access point, usually known as AP, is a device that creates a wireless local area network, or WLAN. In this project the AP is not configured for sharing or projecting an internet connection (bridge), but for transferring data as files resulting from the installed sensors.

The Raspberry Pi can be used as a wireless access point [12], providing a stable and standalone network connection. This can be done using the inbuilt wireless features of the Raspberry Pi 3 or by using a suitable USB wireless dongle that supports access points. In this project inbuilt features of the Raspberry Pi 3 are used.

To work as an access point, the Raspberry Pi does need to have access point software installed, along with DHCP server software to provide connecting devices with a network address.

Firstly, to create an access point is necessary to have DNSMasq and HostAPD installed [12]. To accomplish the installation, type in the command line:

```
sudo apt install dnsmasq hostapd
```

Since the configuration files are not ready yet, it is necessary to stop the new software as follows:

```
sudo systemctl stop dnsmasq  
sudo systemctl stop hostapd
```

At this point, the Raspberry IP address assigned to the wireless port needs to be a static IP to act as a server. It is assumed that the wireless device used is *wlan0*.

To configure the static IP address, the *dhcpcd* configuration file must be edited with the command:

```
sudo nano /etc/dhcpcd.conf
```

Once the file is opened, introduces the interface and the static IP address assigned to the server.

```
interface wlan0  
static ip_address=192.168.4.1/24  
nohook wpa_supplicant
```

Now restarts the *dhcpcd* daemon and set up the new *wlan0* configuration:

```
sudo service dhcpcd restart
```

The DHCP service is provided by *dnsmasq*. By default, the configuration file contains a lot of information that is not needed. It is created a new file and edited the configuration by following:

```
sudo mv /etc/dnsmasq.conf /etc/dnsmasq.conf.orig  
sudo nano /etc/dnsmasq.conf
```

The DHCP service needs to provide a different IP to each device that is connected to the server, editing the file adding:

```
interface=wlan0          # Use the require wireless interface - usually  
wlan0  
dhcp-range=192.168.4.2,192.168.4.20,255.255.255.0,24h
```

For *wlan0*, at this point, it provides an IP address between 192.168.4.2 and 192.168.4.20, with a lease time of 24 hours. The IP 192.168.4.1 is reserved for the server, as it is configured.

The *dnsmasq* is configured, but there are many more options. Start and reload *dnsmasq* to use the updated configuration:

```
sudo systemctl start dnsmasq  
sudo systemctl reload dnsmasq
```

It is needed to edit the hostapd configuration file, located at /etc/hostapd/hostapd.conf, to add the various parameters for your wireless network. After the initial install, this will be a new/empty file.

Furthermore, it is required to configure the host software (hostapd) by editing the hostapd.conf file located at etc/hostapd/hostapd.conf :

```
sudo nano /etc/hostapd/hostapd.conf
```

Add various parameters for the wireless network. Initially, the file is empty:

```
interface=wlan0  
driver=nl80211  
ssid=RaspiWifi  
hw_mode=g  
channel=7  
wmm_enabled=0  
macaddr_acl=0  
auth_algs=1  
ignore_broadcast_ssid=0  
wpa=2  
wpa_passphrase=polplanastfg  
wpa_key_mgmt=WPA-PSK  
wpa_pairwise=TKIP  
rsn_pairwise=CCMP
```

As seen in the code lines it is assumed that channel 7 is used, with a network name (SSID) and a password (wpa_passphrase) that the user chooses. It is also used 2.4 GHz mode (hw_mode) but can be changed with other values. Possible values for hw_mode:

- a = IEEE 802.11a (5 GHz)
- b = IEEE 802.11b (2.4 GHz)
- g = IEEE 802.11g (2.4 GHz)
- ad = IEEE 802.11ad (60 GHz)

Proceed to tell the system where to find this configuration file, locate the line #DAEMON_CONF and replace it with the corresponding path:

```
sudo nano /etc/default/hostapd  
DAEMON_CONF="/etc/hostapd/hostapd.conf"
```

The last action is to enable and start the hostapd. It is important to unmask the hostapd to prevent future errors.

```
sudo systemctl unmask hostapd  
sudo systemctl enable hostapd  
sudo systemctl start hostapd
```

Finally, the access point is created and providing a stable wireless standalone network. Now the user can connect with Raspberry Pi via Wi-Fi searching the SSID configured with any device and introducing the correct password.

4.4. How to connect with a PC or another device

Once the AP is configured and the Wi-Fi is visible for other users any authorised device can connect with the datalogger. In this section, the method used establishes the connection is explained.

Firstly, an external program is required. This software is PuTTY. PuTTY is an SSH and telnet

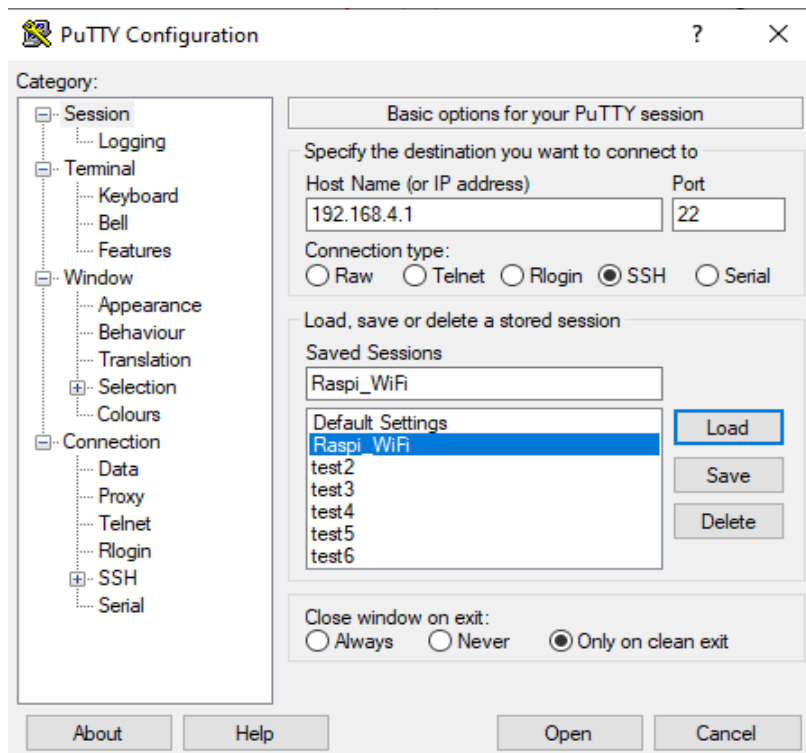
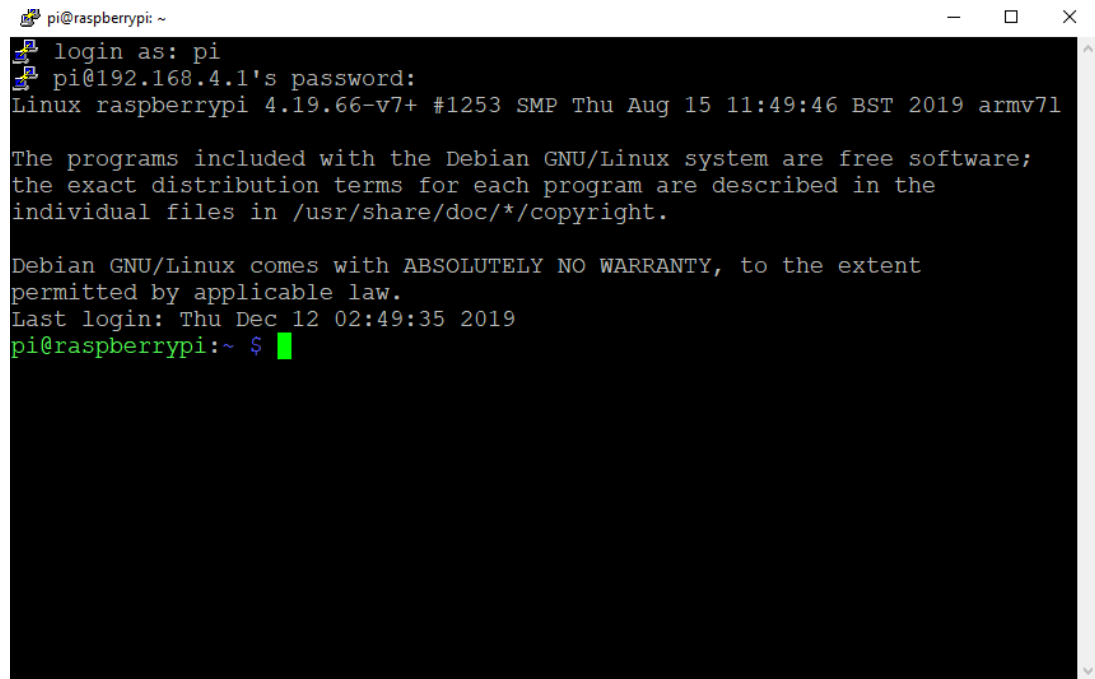


Figure 12. PuTTY configuration window. Source: own.

client [13]. Afterwards, a Wi-Fi connection is needed. The SSID and the password are detailed in section 4.3. The next step is to lunch PuTTY. Then, introduce the static ID created above as shown in Figure 12. Finally, the *open* button is pressed to initiate the connection.

Next, a new command window appears, and the user and password of the Raspberry PI is required (see Figure 13).



```
pi@raspberrypi: ~
login as: pi
pi@192.168.4.1's password:
Linux raspberrypi 4.19.66-v7+ #1253 SMP Thu Aug 15 11:49:46 BST 2019 armv7l

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Thu Dec 12 02:49:35 2019
pi@raspberrypi:~ $
```

Figure 13. PuTTY command window. Source: own.

Now the connection is stablished and the device can be remotely controlled.

4.5. Schematic of communication

The device has wireless communication to facilitate interaction between the client and the datalogger. The client can send commands to the server and it executes them and/or responds to the client. This is achieved with the configuration of the sockets for the Wi-Fi network. Below a simple scheme of the communications is shown.

There are two types of functions: unidirectional and bidirectional. As it is shown in Figure 14, the *start/stop* functions and the *configuration* functions, which are unidirectional. The commands are sent by the client and the server executes them and no feedback is sent.

On the other hand, the *transfer* function is bidirectional. When the transfer command is executed by the client, the logged files in the server are immediately sent to the client, so feedback is done by the server.

Next, a simple block diagram to represent the information flow system.

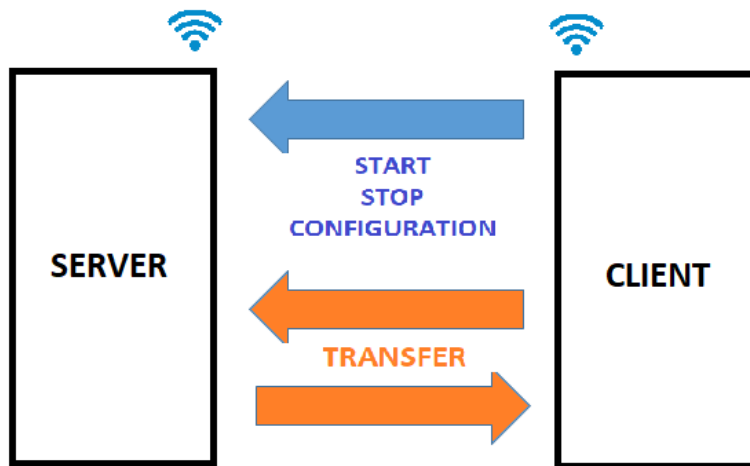


Figure 14. Schematic of communication. Source: own.

4.6. Default modules

Creating an own code is a laborious task, even more without the help of pre-installed modules in Python libraries. Next, the default Python modules used for this project will be described. The main functions used by the device are also detailed.

4.6.1. Socket module

Sockets [14] allow communication between two different processes on the same or different machines. It is a way to talk to other computers using standard Unix file descriptors.

A Unix Socket [14] is used in a client-server application framework. A server - Raspberry PI - is a process that performs some functions on request from a client – users -. Most of the application-level protocols like FTP, SMTP, and POP3 make use of sockets to establish the connection between client and server and then for exchanging data.

There are four types of sockets available to the users, but in this project, only one is used [15]:

- Stream Sockets – Delivery in a networked environment is guaranteed. If three items "A, B, C" are send through the stream, data arrives in the same order – "A, B, C". These sockets use TCP (Transmission Control Protocol) for data transmission. If delivery is impossible, the sender receives an error indicator. Data records do not have any boundaries.

As a Python module [15], it has several internal functions and objects, but only the main ones for the project are briefly detailed.

- **socket.socket**(family=AF_INET, type=SOCK_STREAM, proto=0, fileno=None): Create a new socket using the given address family, socket type and protocol number.
- **socket.bind**(address): Bind the socket to *address*. The socket must not already be bound.
- **socket.listen**([backlog]): Enable a server to accept connections. If the backlog is specified, it must be at least 0 (if it is lower, it is set to 0); it specifies the number of unaccepted connections that the system will allow before refusing new connections. If not specified, a default reasonable value is chosen.
- **socket.accept**(): Accept a connection. The socket must be bound to an address and listening for connections. The return value is pair (*conn*, *address*) where *conn* is a new socket object usable to send and receive data on the connection, and *address* is the address bound to the socket on the other end of the connection.
- **socket.recv**(bufsize[, flags]): Receive data from the socket. The return value is a byte object representing the data received. The maximum amount of data to be received at once is specified by *bufsize*.
- **socket.sendfile**(file, offset=0, count=None): Send a file until EOF is reached by using high-performance *os.sendfile* and return the total number of bytes which were sent. *file* must be a regular file object opened in binary mode. If *os.sendfile* is not available (e.g. Windows) or *file* is not a regular file *send()* will be used instead. *offset* tells from where to start reading the file. If specified, *count* is the total number of bytes to transmit as opposed to sending the file until EOF is reached. File position is updated on return or also in case of error in which case *file.tell()* can be used to figure out the number of bytes that were sent. The socket must be of *SOCK_STREAM* type. Non-blocking sockets are not supported.
- **socket.settimeout**(value): Set a timeout on blocking socket operations.
- **socket.close**(): Close a socket file descriptor.

4.6.2. Threading module

A thread is a separate flow of execution [16]. Python threading allows having different parts of a program running concurrently and can simplify the design. For this project, two main threads are created: the first one controls all the communications running the socket module. The second one is created for the process of logging all data of the sensor breakout. Main functions and used modules are briefly detailed [16]:

- Class threading. **Thread**(group=None, target=None, name=None, args=(), kwargs={}, *, daemon=None): This constructor should always be called with keyword arguments. Arguments are: group should be None; reserved for future extension when a ThreadGroup class is implemented. *target* is the callable object to be invoked by the *run()* method. Defaults to *None*, meaning nothing is called. *name* is the thread name. By default, a unique name is constructed of the form “*Thread-N*” where *N* is a small decimal number. *args* is the argument tuple for the target invocation. Defaults to (). *kwargs* is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If the subclass overrides the constructor, it must make sure to invoke the base class constructor (*Thread.__init__()*) before doing anything else to the thread.

- Thread.**start**(): Start the thread's activity. It must be called at most once per thread object. It arranges for the object's *run()* method to be invoked in a separate thread of control.

This method will raise a *RuntimeError* if called more than once on the same thread object.

- Thread.**join**(timeout=None): Wait until the thread terminates. This blocks the calling thread until the thread whose *join()* method is called terminates – either normally or through an unhandled exception – or until the optional timeout occurs. When the timeout argument is present and not None, it should be a floating-point number specifying a timeout for the operation in seconds (or fractions thereof). As *join()* always return *None*, you must call *is_alive()* after *join()* to decide whether a timeout happened – if the thread is still alive, the *join()* call timed out.

4.6.3. Time module

This module [17] provides various time-related functions. In this project, it is used to calculate the time of every sample logged for the post-processing of data. The functions of the used time module are summarised in this section [17].

An explanation of some terminology and conventions is in order.

- The *epoch* is the point where the time starts and is platform dependent. For Unix, the epoch is January 1, 1970, 00:00:00 (UTC).
- The term *seconds since the epoch* refers to the total number of elapsed seconds since the epoch, typically excluding *leap seconds*.
- The functions in this module may not handle dates and times before the epoch or far in the future. The cut-off point in the future is determined by the C library; for 32-bit systems, it is typically in 2038.
- time.**time**(): Return the time in seconds since the *epoch* as a floating-point number.

- `time.sleep()` suspends execution for the given number of seconds. The argument may be a floating-point number to indicate a more precise sleep time.

4.6.4. Datetime module

The *datetime* module [18] supplies classes for manipulating *dates* and *times* in both simple and complex ways. While date and time arithmetic are supported, the focus of the implementation is on efficient attribute extraction for output formatting and manipulation.

There are two kinds of date and time objects: “naive” and “aware”.

An *aware* object has sufficient knowledge of applicable algorithmic and political time adjustments, such as time zone and daylight-saving time information, to locate itself relative to other aware objects. An aware object is used to represent a specific moment in time that is not open to interpretation.

A *naive* object does not contain enough information to unambiguously locate itself relative to other date/time objects. Whether a naive object represents Coordinated Universal Time (UTC), local time, or time in some other timezone is purely up to the program, just like it is up to the program whether a particular number represents metres, miles, or mass. Naive objects are easy to understand and to work with, at the cost of ignoring some aspects of reality.

- `datetime.now([tz])`: Return the current local date and time. If optional argument *tz* is *None* or not specified, this is like *today()*, but, if possible, supplies more precision than can be gotten from going through a *time.time()* timestamp (for example, this may be possible on platforms supplying the C *gettimeofday()* function).

If *tz* is not *None*, it must be an instance of a *tzinfo* subclass, and the current date and time are converted to *tz*’s time zone. In this case the result is equivalent to `tz.fromutc(datetime.utcnow().replace(tzinfo=tz))`.

4.7. Code

As mentioned before, code programming is an essential aspect of the performing of the datalogger. The purpose of this program is to manage all the features related to the control and distribution of the data. The program is clearly separated by two scripts: first one is for server control. The second one is for client control.

These scripts are public and can be found in a repository created in Github. GitHub [19] is a global company that provides hosting for software development version control using Git. It has more than 100 million repositories hosted.

Briefly, the functions are listed in the Figure 15 to have a more global view of the datalogger:

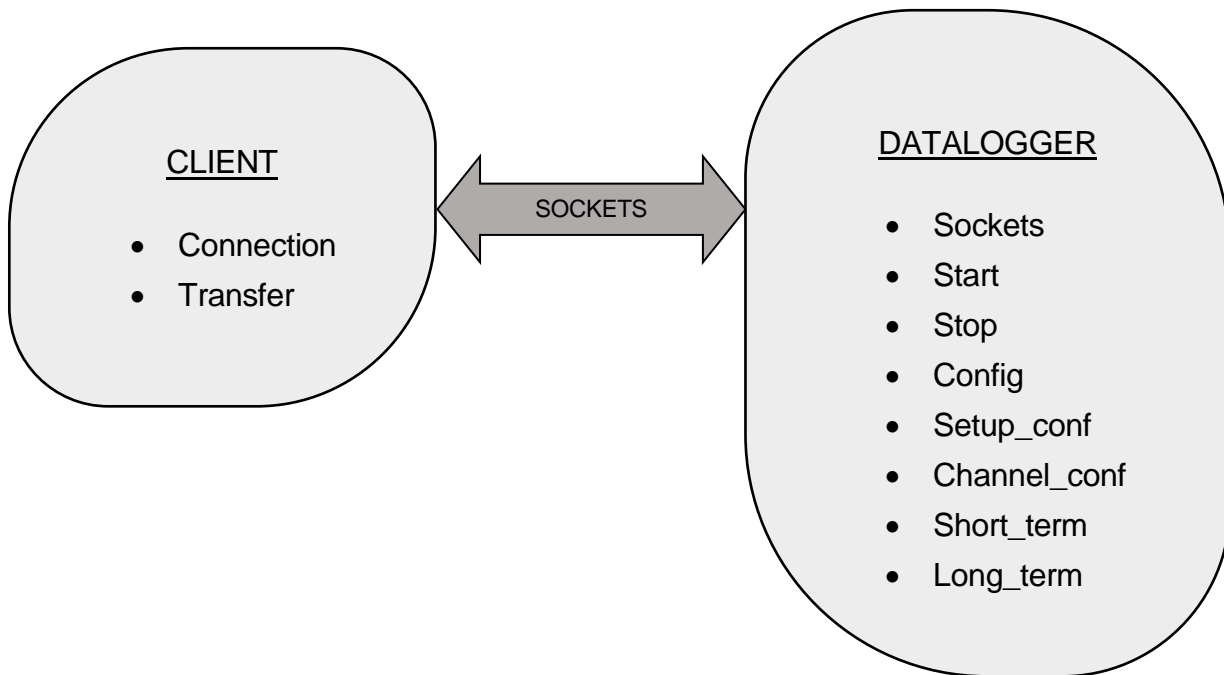


Figure 15. Client and Server scripts. Source: own.

Next, the operation of the functions developed for the datalogger application is explained.

4.7.1. Class Datalogger

This class is used for server script, located on the Raspberry, since it must collect the data.

For reader easy understanding, the main variables are detailed:

- Channels: these variables mean the channel/sensor that the user wants to use.
 - CHANNEL_1: to use only sensor breakout 1
 - CHANNEL_2: to use only sensor breakout 2
 - CHANNEL_12: to use both sensor breakouts
- Setup: these variables mean the configuration of datalogging the user wants to use.
 - LONG_TERM: this configuration allows leaving the device logging indefinitely unless interrupted by the user.
 - SHORT_TERM: this configuration allows programming a determined number of samples that the user wants to log.

4.7.1.1. __init__

The function "`__init__`" is a reserved method in python classes [20]. It is known as a constructor in object-oriented concepts. This method called when an object is created from the class and

it allows the class to initialize the attributes of a class.

In this case, the user can give initial values to the variables *setup* and *channel*, in this order. If no values are given, the default values are initialized:

- Setup = SHORT_TERM
- Channel = CHANNEL_12

It can be found different blocks of initialization objects:

- Sockets block, where the initial configuration for the server-client socket communication is initialized in a thread.
- GPIO block, where the two sensor breakouts are pre-configured with initial default values.

4.7.1.2. Sockets

This function is called by the communication thread to create a loop that can be waiting for new commands from the client. When a command is received, it is detected and processed.

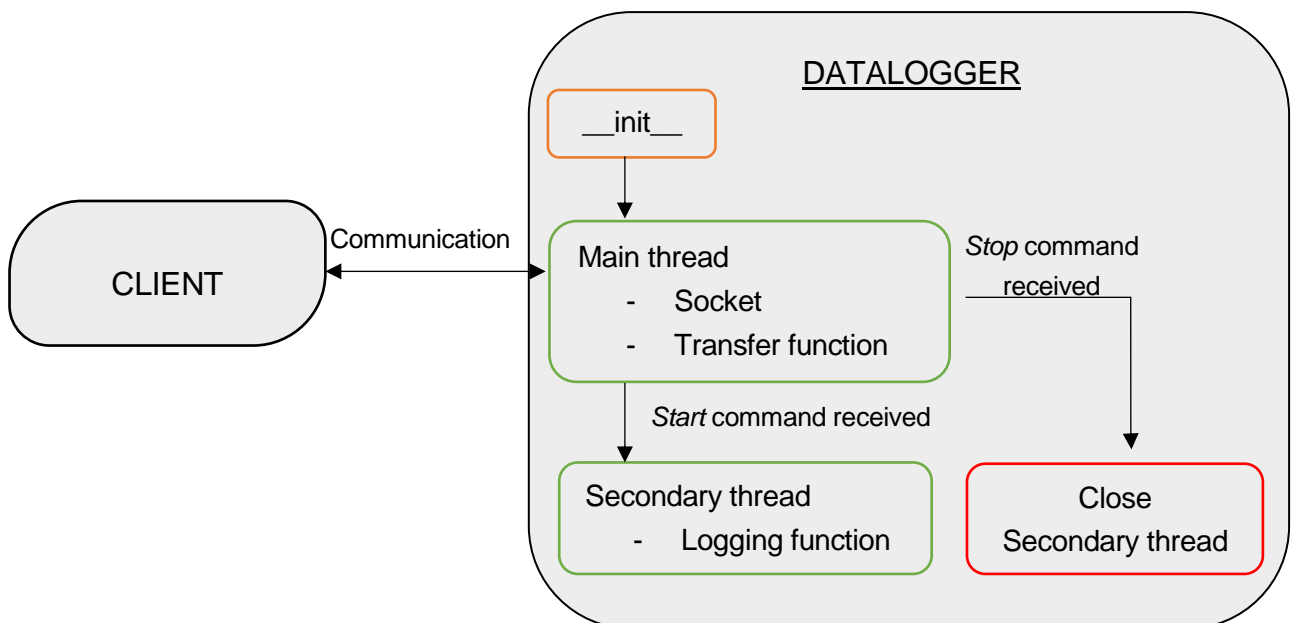


Figure 16. Block diagram for threads. Source: own.

If the received data is a correct command, the program executes it. Otherwise, an error message is printed on the screen. The following commands are accepted:

- Start: the function *start* is launched.
- Stop: the function *stop* is launched.
- Transfer: once the data is stored in a file, the function *transfer* is launched.
- Close all: close the communication between server and client.

As shown in Figure 17, a flow diagram describes the process for sockets connection and communication between server and client.

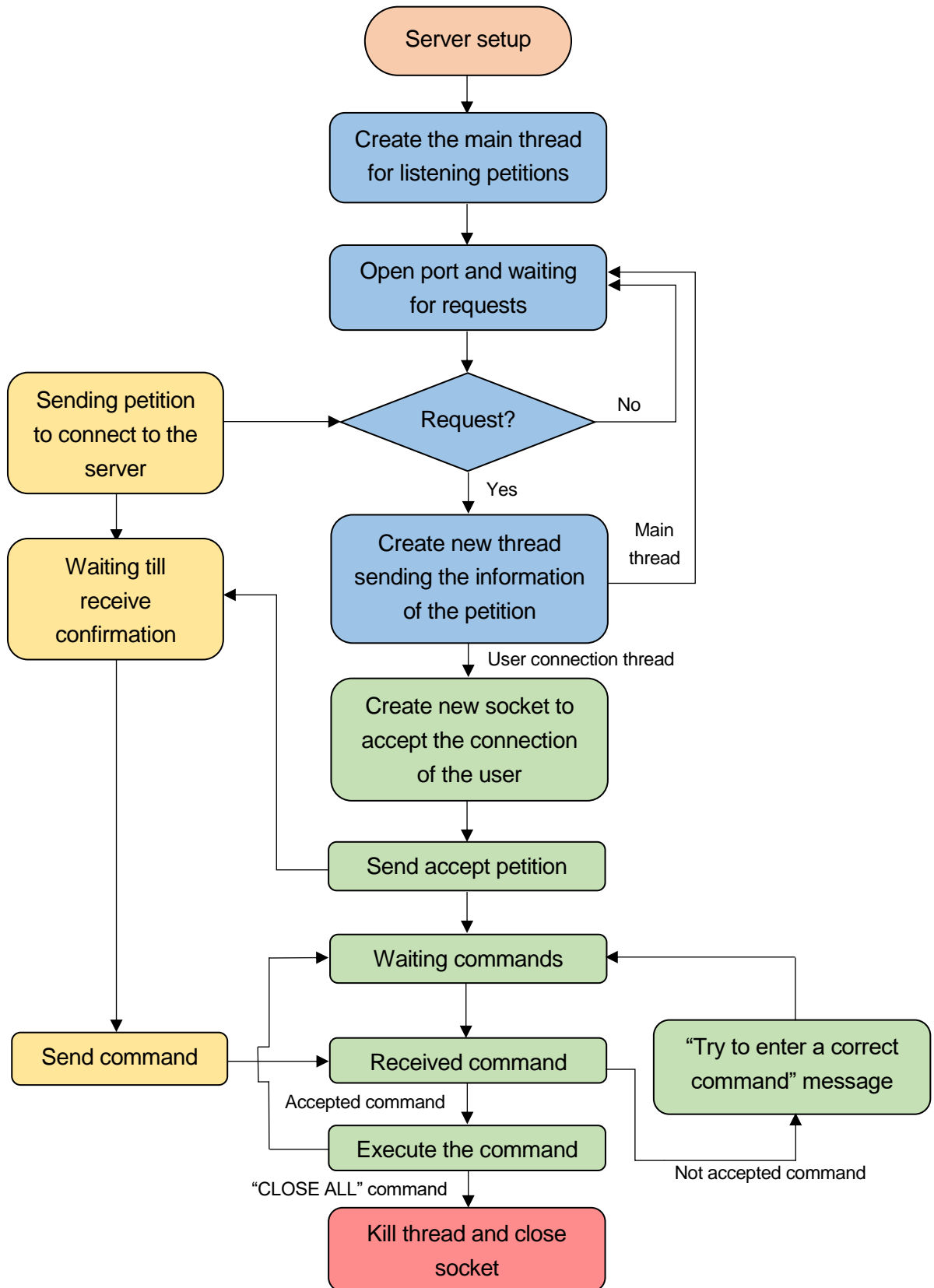


Figure 17. Flowchart for socket process. Source: own.

4.7.1.3. Start/stop

These functions are very simple. They are briefly explained:

- Start: when this function is called, it creates a new thread named *process*, where the datalogging process is started. This thread is created by the *threadloopstartstop* module described in section 4.7.3. The *start* function of this module is called. The object *mode* changes to *ON*.
- Stop: when this function is called, it calls the function *stop* from the *threadloopstartstop* module to force the datalogging process to stop. The object *mode* changes to *OFF*.

4.7.1.4. Sample_conf

This function allows for changing the number of samples. The number of samples needs to be given as an argument. The argument must be an integer type number.

4.7.1.5. Setup_conf

This function is called if a different setup configuration is required. The setup needs to be given as an argument. The argument must be a *setup* variable described in point 4.7.1.

4.7.1.6. Channel_conf

The same as the *setup_conf*, this function is called if a different channel configuration is required. The channel needs to be given as an argument. The argument must be a *channel* variable described in point 4.7.1.

4.7.1.7. Short_term / Long_term

The *short_term* function is used if a specific number of samples are required, while *long_term* function is called if the user requires a long, or indefinitely, logging time. It is important to configure the datalogger before to start any function. The function chosen will start with the last settings configured or with default values if no pre-settings were configured. No arguments needed. The default value for *channel* and *setup* variable is detailed in point 4.7.1.1.

The functions log the values of voltage and current for each sensor breakout. This data is saved in individual text files, with values of current and voltage for each sensor breakout. Into the file, it is shown a column timestamp and a column with the measured value (see Figure 18).

```

z.voltage_ch1: Bloc de notas
Archivo Edición Formato Ver Ayuda
% Timestamp, Voltage ch1 (V)
1578753358.3932428 ; 7.784
1578753358.4196925 ; 7.784
1578753358.4356434 ; 7.784
1578753358.4546452 ; 7.784
1578753358.4743788 ; 7.784
1578753358.4902604 ; 7.784
1578753358.503816 ; 7.784
1578753358.5169034 ; 7.784
1578753358.5304644 ; 7.784
1578753358.5436766 ; 7.784
1578753358.5576088 ; 7.78
1578753358.5713086 ; 7.78
1578753358.5834801 ; 7.784
1578753358.5963748 ; 7.78
1578753358.6095278 ; 7.78
1578753358.620448 ; 7.78
1578753358.6358392 ; 7.78
1578753358.6497087 ; 7.784

```

Figure 18. Format of the datalogger file. Source: own.

It is of great relevance to notice that when the two sensor breakouts are logging data, firstly, the voltage value is saved and after a short period of time, the current value is saved. Therefore, a time lag is generated between both samples.

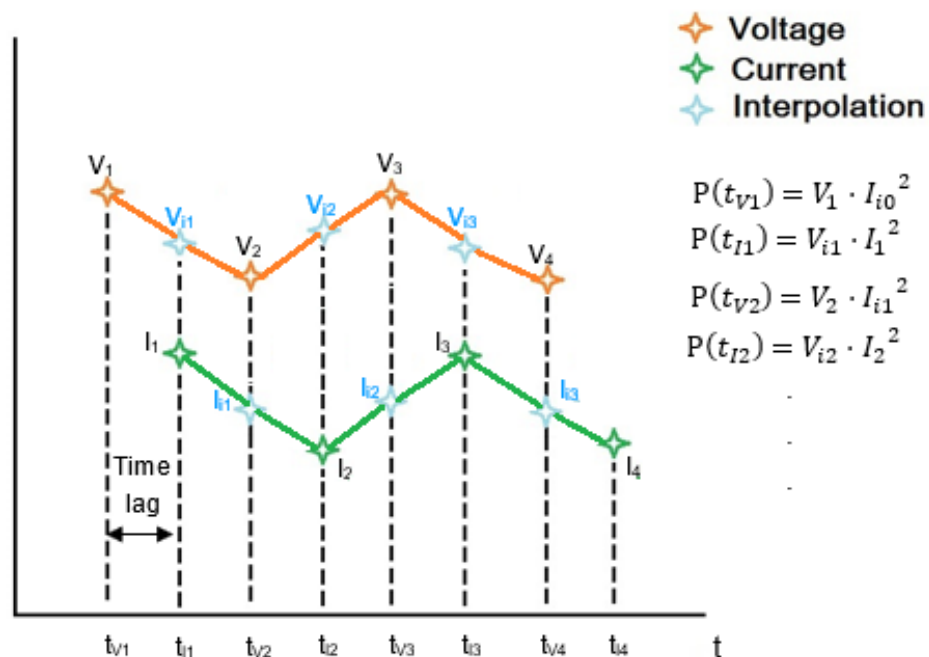


Figure 19. Example of chart for samples logged and interpolated. Source: own.

This issue is solved by an external interpolation Matlab script, as described in section 4.7.4.

4.7.1.8. Transfer

This function allows transferring the files created by the logging function using a compressed file in a zip format. No arguments are needed.

The way this function sends the file is the following:

First, a block of bytes is read from the zip file. The block size is *CHUNK_SIZE*, defined in the `__init__` function. Afterward, these bytes are sent and the client receives them. The client transfer function is detailed in point 4.7.2.3.

Finally, the two previous steps are repeated until all bytes are read and sent.

It is not necessary to stop the logging function. All the data saved up to that moment will be transferred and the logging function will continue saving data. It is important to note that the file does not close, therefore the next file transferred will contain the previously sent data. To start a new file, the logging function must be stopped.

4.7.2. Class Client

This class is used for client script, located on the user device.

4.7.2.1. `__init__`

In this case, no initial arguments are required. It can be found different blocks of initialization objects:

- Sockets block where the initial configuration for the server-client socket communication is initialized. See that the IP used for the connection is the pre-configured static IP detailed in point 4.3.
- Transfer block where the *CHUNK_SIZE* variable is initialized.

4.7.2.2. Connection

The purpose of the function is to allow the user to communicate with the server by sending messages. It allows any command; the server is responsible to select the messages and decide if are accepted or not. Two determined commands are defined:

- *Transfer*: if this command is called it is sent to the server to execute the *Transfer* function detailed in point 4.7.1.8 and launch the function *transferClient* defined in point 4.7.2.3.
- *Close all*: if this command is called, it is sent to the server to close the socket server connection and it closes the socket client connection.

4.7.2.3. TransferClient

This function also receives the file sent by the server in blocks. The block sent by the server is received and saved in a new file since all blocks of bytes are received. These files are compressed in a zip file.

4.7.3. Class threadloopstartstop

The *threadloopstartstop* class is a secondary class created to provide new threads for the main class *-Datalogger-*.

This class is an inherited class from the general *threading* class. Inheritance is the capability of one class to derive or inherit the properties from some other class.

For this class, five arguments are required to initialize. Briefly, the arguments are described:

- Interval: Time argument. This parameter implements how often the function selected is wanted to be executed.
- Default: Boolean argument.
 - True: thread activated
 - False: thread not activated

In this particular case, the argument is *False* due to the thread starts when the *start* function defined in point 4.7.1.3 is called by the user.

- Function: for this argument, the name of the function wanted to be executed is required.
- Args: this parameter is the arguments needed for the function selected. If no arguments are required for the function, then this parameter shall not be filled.
- Kwargs: key-words arguments. Argument not required.

Four simple functions are implemented to perform basic control of the created threads. These functions are the following:

- Stop: it allows stop the thread buckle so it cannot execute the function. The thread is still alive.
- Restart: it allows initiate the thread and start executing the function.
- Shutdown: once the thread is stopped, it can be closed with this function. The thread

is killed.

- Isstopped: it shows information about the thread state. A Boolean is shown.
 - If False: the thread is running
 - If True: the thread is stopped.

4.7.4. Interpolation Matlab script

As it is commented in section 4.7.1.7 a time lag is generated between the samples of both sensor breakouts. Once the user has transferred the files, this Matlab script loads the files. Due to the format of the files, the subsequent evaluation of the data is not difficult.

The electronic circuit shown in Figure 20 is composed by a 100 kΩ potentiometer and a 0,1 kΩ resistor to create a real load.

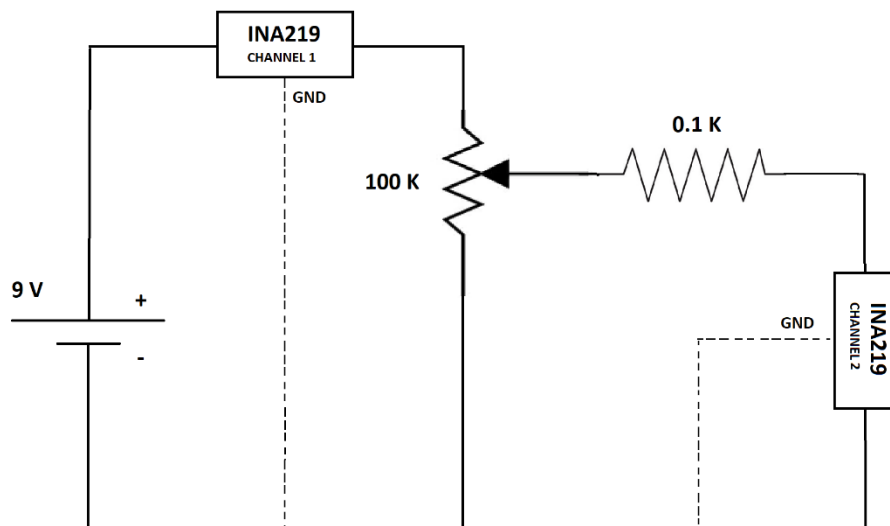


Figure 20. Matlab command window. Source: own.

First, the variables are separated in voltage and current variables, and time variables, for each channel.

Afterwards, the voltage, the current and the time are interpolated so power calculation can be done with voltage and current in the same space of time. Then the energy consumption and efficiency are calculated and displayed in the command window (Figure 21).

```
Command Window
New to MATLAB? See resources for Getting Started.

Loading files...
First time in v1: 11-Jan-2020 14:35:58
First time in c1: 11-Jan-2020 14:35:58
First time in v2: 11-Jan-2020 14:35:58
First time in c2: 11-Jan-2020 14:35:58
Lower integration limit: 11-Jan-2020 14:35:58
Last time in v1: 11-Jan-2020 14:36:03
Last time in c1: 11-Jan-2020 14:36:03
Last time in v2: 11-Jan-2020 14:36:03
Last time in c2: 11-Jan-2020 14:36:03
Upper integration limit: 11-Jan-2020 14:36:03
Energy balance ch1: E1=5.1233
Energy balance ch2: E2=0
Difference: |E1|-|E2|=5.1233
fx >>
```

Figure 21. Matlab command window. Source: own.

Finally, as it is shown in Figure 22 the voltage and current data are plotted in graphics for both channels.

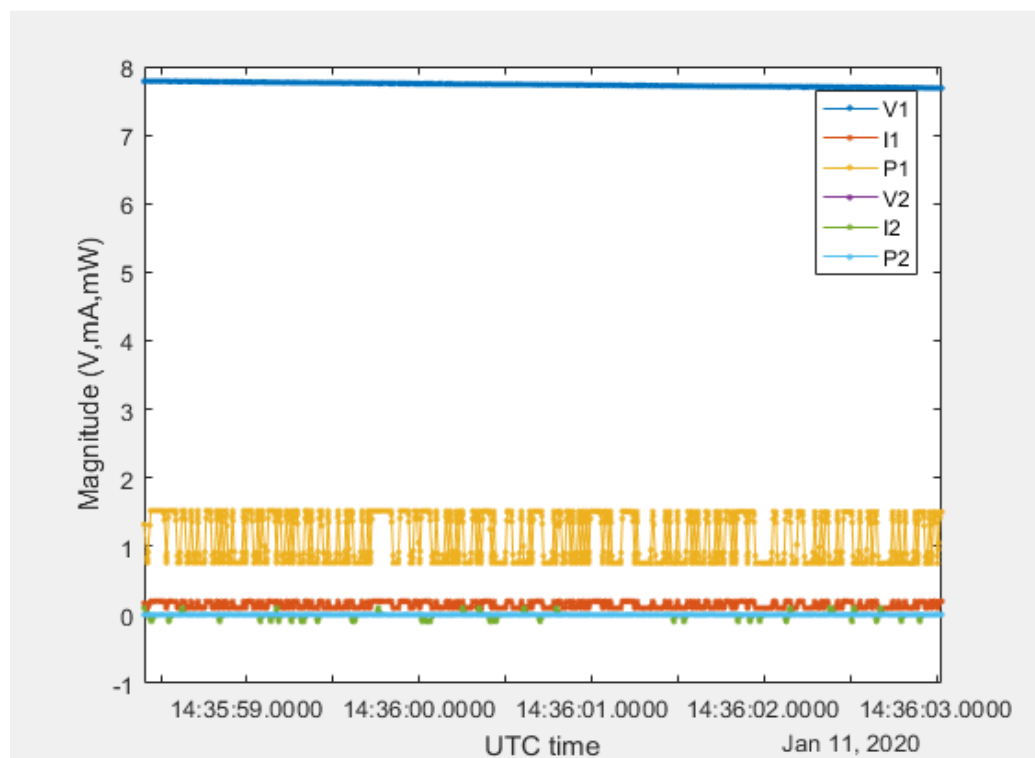


Figure 22. Graphics window for voltage, current and power. Source: own.

5. Test and validation

Once the program has been implemented, it is mandatory to perform several tests to validate that all functions work properly. Individual tests are performed for each function in the class *Datalogger*.

In the first place, logging functions and configure functions are tested. Afterwards, communication functions are tested. Finally, multi-function test is performed.

5.1. Short_term test

Using the code shown in Figure 23, *short_term* function is tested.

```
454 if __name__ == "__main__":  
455     try:  
456         t=Datalogger(SHORT_TERM, CHANNEL_12)  
457         t.start()  
458     except KeyboardInterrupt:  
459         exit()  
460
```

Figure 23. Commands for *short_term* function. Source: own.

As shown in Figure 24, the datalogger launches the *short_term* function and the default number of samples are logged. The results are as expected.

```
pi@raspberrypi:~/Desktop/Code/final $ python3 DATALOGGER.py  
DATALOGGING...  
LOGGED 1000 samples  
█
```

Figure 24. Results for *short_term* function. Source: own.

5.2. Long_term test

Using the code shown in Figure 25, *long_term* function is tested.

```
454 if __name__ == "__main__":  
455     try:  
456         t=Datalogger(LONG_TERM, CHANNEL_12)  
457         t.start()  
458         time.sleep(5)  
459         t.stop()  
460  
461  
462     except KeyboardInterrupt:
```

Figure 25. Commands for *long_term* function. Source: own.

As shown in Figure 26, the datalogger launches the *long_term* function and starts logging samples for five seconds, then the command *stop* is launched and the datalogger is stopped. The results are as expected.

```
pi@raspberrypi:~/Desktop/Code/final $ python3 DATALOGGER.py  
DATALOGGING...  
LOGGED 341 samples  
pi@raspberrypi:~/Desktop/Code/final $
```

Figure 26. Results for *long_term* function. Source: own.

5.3. Start / Stop test

Using the code shown in Figure 27, *start* and *stop* functions are tested. It is supposed the application starts and, after 5 second stops logging.

```
449  
450 if __name__ == "__main__":  
451     try:  
452         t=Datalogger(SHORT_TERM, CHANNEL_12)  
453         t.start()  
454         time.sleep(5)  
455         t.stop()  
456  
457  
458     except KeyboardInterrupt:  
459         exit()
```

Figure 27. Commands for *start/stop* functions. Source: own.

As shown in Figure 27, the datalogger starts logging and after 5 seconds *stop* function is activated and the datalogger stops, showing on the screen the samples logged, Figure 28. The results are as expected.

```
pi@raspberrypi:~/Desktop/Code/final $ python3 DATALOGGER.py
DATALOGGING...
LOGGED 382 samples
pi@raspberrypi:~/Desktop/Code/final $
```

Figure 28. Results for *start/stop* function. Source: own.

5.4. Sample_conf test

Using the code shown in Figure 29, *config* function is tested. This test is done in SHORT_TERM mode.

```
453
454 if __name__ == "__main__":
455     try:
456         t=Datalogger(SHORT_TERM, CHANNEL_12)
457         t.sample_conf(555)
458         t.start()
459
460
461     except KeyboardInterrupt:
462         exit()
```

Figure 29. Commands for *sample_conf* function. Source: own.

As shown in Figure 30, the results are printed on the screen. Afterwards, *config* function changed the number of samples and logged them. The results are as expected.

```
pi@raspberrypi:~/Desktop/Code/final $ python3 DATALOGGER.py
New samples: 555
DATALOGGING...
LOGGED 555 samples
```

Figure 30. Results for *sample_conf* function. Source: own.

5.5. Sockets and transfer test

This *socket* function allows the communication between the datalogger and the user. To validate it works properly, a communication test is performed. The test consists of sending commands from the user to the datalogger and execute them.

```
pi@raspberrypi:~/Desktop/Code/final $ python3 CLIENT.py
tcpClientA: Enter message to continue/ Enter exit:start
tcpClientA: Enter message to continue/ Enter exit:stop
tcpClientA: Enter message to continue/ Enter exit:transfer
File has been received!
tcpClientA: Enter message to continue/ Enter exit:
```

Figure 31. Commands from user terminal. Source: own.

As it is shown in Figure 31, several commands are sent to the datalogger which are executed, as shown in Figure 32. The results are as expected.

```
pi@raspberrypi:~/Desktop/Code/final $ python3 DATALOGGER.py
Multithreaded Python server : Waiting for connections from TCP clients...
DATALOGGING...
LOGGED 921 samples
File has been sent

```

Figure 32. Results for user's commands. Source: own.

For *transfer* function validation, the files logged in *socket* function test are used. Sending "transfer" command from the client to the datalogger to proof the files are transferred properly. The results are as expected (see Figure 31 and Figure 32).

5.6. Multi-function test

This test pretends to simulate the operation conditions executing several commands. As shown in Figure 33, firstly the datalogger is initialized to start the long-term configuration and both sensor breakouts working. After ten seconds, the channel is changed and only sensor breakout 2 is working. Afterwards, the default number of samples is changed to 2500 samples and the setup configuration is changed to the short term.

```

444
445 if __name__ == "__main__":
446     try:
447         t=Datalogger(LONG_TERM, CHANNEL_12)
448         t.start()
449         time.sleep(10)
450         t.channel_conf(CHANNEL_2)
451         t.start()
452         time.sleep(5)
453         t.sample_conf(2500)
454         t.setup_conf(SHORT_TERM)
455         t.start()
456
457
458     except KeyboardInterrupt:
459         exit()

```

Figure 33. Commands for *multi- function* test. Source: own.

As shown in Figure 34, the server and the client are successfully connected and the test starts executing all the commands. The results are as expected.

```

pi@raspberrypi:~/Desktop/Code/final $ python3 DATALOGGER.py
Multithreaded Python server : Waiting for connections from TCP clients...
DATALOGGING...
LOGGED 732 samples
Changed to channel 2
DATALOGGING...
LOGGED 486 samples
New samples: 2500
Setup changed to SHORT_TERM
DATALOGGING...
LOGGED 2500 samples

```

Figure 34. Results for *multi- function* test. Source: own.

6. Difficulties and solutions

When a project is initiated, it is not always developed as expected. Difficulties arise during the development of the project that must be analysed and find the best solution or alternative to continue with the project.

In this project I have had to face different difficulties. The problems and solutions chosen are shown.

- The first difficulty was the configuration of the access point. Firstly, to develop the AP a Raspberry Pi 2 and a Wi-Fi modem were chosen. The idea was to configure the Wi-Fi as an access point due to the Raspberry Pi 2 did not have a wireless antenna. Advanced knowledge of informatics was required to configure the modem to the Raspberry Pi.

The solution chosen was to change to a Raspberry Pi 3 Model B with an integrated wireless antenna and do the configuration shown in section 4.3.

- After that, an issue appears while developing the datalogger code. The transfer time for the data files was too slow due to several processes were working at the same time. Multiple threads share the same processor and this fact causes saturation.

In the first instance, the *multiprocessing* module was an alternative due to each process uses independent processors. The problem with this module was that the process for sockets could not communicate with the logger process, hence wireless command did not work.

Finally, the best solution was to upgrade and optimize the transfer code sending files by block instead of byte for byte.

- Finally, one of the INA219 breakout stopped working. A new INA219 breakout was bought and replaced the broken one.

7. Budget

It is important to collect the used material and the spent hours in this project to have an economic view of the project. For the hours involved in the project development a price of 50 €/h, as an engineer, has been considered.

<i>Item</i>	<i>Quantity</i>	<i>Price</i>
<i>Raspberry Pi 3 model B</i>	1	40 €
<i>Micro SD</i>	1	5 €
<i>Mini wireless keyboard</i>	1	25 €
<i>INA219 sensor</i>	3	30€
<i>Bread Board</i>	1	1 €
<i>GPIO pin connector</i>	1	0.5 €
<i>PC amortization</i>	-	65 €

<i>Month</i>	<i>Description</i>	<i>Hours</i>	<i>Price</i>
<i>September</i>	Research and set up Raspberry Pi	55	2.750€
<i>October</i>	Soldering work and programming	75	3.750 €
<i>November</i>	Programming and document	65	3.250 €
<i>December</i>	Programming and document	60	3.000 €
<i>January</i>	Tests and document	55	2.750 €
		310	15.500 €
Total			15.666,5€

Table 4. Budget. Source: own.

8. Environmental impact

Electrical and electronic devices contain dangerous contaminants that pollute the environment and pose a high health risk, therefore the importance of their proper disposal [21].

These wastes contain toxic substances and heavy metals that when in contact with the earth, water or air put the environment at risk.

Recycling electrical and electronic equipment by taking them to an authorized manager, not only prevents the contamination that they directly produce but also, by recovering the materials, contributes to a much more supportive economic cycle with our planet.

Some possible solutions could be:

- Incorporate responsible consumption that includes the recycling of electronic equipment.
- Reduce the generation of electronic waste through responsible purchasing and good maintenance.
- Sell or donate the electronic devices that still working.
- Donate broken or old equipment to organizations that repair and reuse it for social purposes.
- Recycle components that cannot be repaired.

The Raspberry Pi used for this project can be reused in multiples ways or used for other projects if it is still working. If not, many solutions are given above.

Notice that all Raspberry Pi models and all components used in this project (wires, protoboards, etc.) comply the RoHs Regulation. All Raspberry Pi products have undergone extensive compliance testing, and copies of the relevant certificates and conformity documents are available on the webpage [22].

The contribution of this project is positive due to it permits to improve and optimize the Internet of Things boards with solar and thermal harvesting, hence it allows to reduce the environmental impact.

Conclusions and future work

In conclusion, it can be mentioned that the main objective of the project has been achieved. A datalogger has been successfully developed and it can acquire and store the current and voltage magnitudes of two electric nodes. Also, a wireless access point has been implemented to perform basic remote-control functions.

Nonetheless, the datalogger can be improved as this is a very new version that only performs most basic functions. In order to improve the device, and as a future work some features could be updated and improved, which are the following:

- In the first place, as the datalogger has only basic remote-control functions, one point to improve would be to implement a more complex function to do the device more versatile and friendly for the user, as for example configuration-setup functions. For the same reason, a time function could be programmed to allow the user to set a logging time.
- In the second place, as commented in section 4.7.1.7, a time lag between samples of the sensors is generated. This issue could be solved by implementing a self-interpolation program in the datalogger to calculate power and efficiency.
- Also, it could be interesting to display all data in real-time, so another future work would be to develop a program that could show graphics on a screen.
- Another interesting feature could be to permit several clients connecting at the same time so they could get the data files. This could be achieved by updating the code with the *select* module, for example.
- Finally, the graphics display could be improved by creating a graphic interface. An interesting option for this is to create a simple application where the configuration could be set up and start/stop button.

In addition to accomplishing the main objectives, the derived objectives and personal objectives of this project - such as improve my programming skills and put into practice all my knowledge learned in the education as an engineer- have been also achieved.

Acknowledgement

I would like to thank the great support and help that my supervisor Manuel Moreno Eguílaz has given me in this project. He has always been available to help me with any problem I had. Without his guidance, it would have been a much harder job. Special thanks to co-supervisor, Álvaro Gómez, for helping me with the interpolation Matlab script.

I would also thank the Polytechnic University of Barcelona for the opportunity to learn all the knowledge required to develop this project.

Of course, I should also thank all those people who have helped me selflessly so I can progress on this project.

Finally, I also would like to thank my family and friends for the support given during these years in my education at the university and making it all easier.

References

- [1] Catalunya, U. P. de, & Upc. (2016, July 11). AMBER UPC. Retrieved December 18, 2019, from <https://amber.upc.edu/es>.
- [2] Raspberry Pi. (2019, December 22). Retrieved September 19, 2019, from https://en.wikipedia.org/wiki/Raspberry_Pi
- [3] Fitzpatrick, J. (2017, October 30). Raspberry Pi hardware. Retrieved September 27, 2019, from <https://www.howtogeek.com/138281/the-htg-guide-to-getting-started-with-raspberry-pi/>.
- [4] Sum, P. E. (2017, December 26). Control GPIO with Python in Raspberry Pi. Retrieved November 25, 2019, from <https://www.programoergosum.com/cursos-online/raspberry-pi/238-control-de-gpio-con-python-en-raspberry-pi/que-es-gpio>.
- [5] INA219's Datasheet. (2015, December 1). Retrieved September 19, 2019, from <http://www.ti.com/lit/ds/symlink/ina219.pdf>.
- [6] INA219 High Side DC Current Sensor Breakout. (2017, July 10). Retrieved December 30, 2019, from <https://core-electronics.com.au/ina219-high-side-dc-current-sensor-breakout-26v-3-2a-max.html>.
- [7] Raspbian. (2017, October 11). Retrieved October 4, 2019, from <https://www.raspberrypi.org/documentation/raspbian/>.
- [8] Download Raspbian for Raspberry Pi. (2019, July 10). Retrieved September 17, 2019, from <https://www.raspberrypi.org/downloads/raspbian/>
- [9] Software *balenaEtcher*. Retrieved from <https://www.balena.io/etcher/>
- [10] Python Package Index (PyPi). (n.d.). Retrieved October 10, 2019, from <https://pypi.org/>.
- [11] chrisb2. (2019, July 9). chrisb2/pi_ina219. Retrieved September 18, 2019, from https://github.com/chrisb2/pi_ina219.
- [12] Setting up a Raspberry Pi as a Wireless Access Point. (2019, August 5). Retrieved September 18, 2019, from <https://www.raspberrypi.org/documentation/configuration/wireless/access-point.md>.
- [13] PuTTY (n.d.). Retrieved January 10, 2020, from <https://www.putty.org/>
- [14] What is a Socket? (n.d.). Retrieved December 5, 2019, from https://www.tutorialspoint.com/unix_sockets/what_is_socket.htm.

- [15] socket - Low-level networking interface. (n.d.). Retrieved December 5, 2019, from <https://docs.python.org/3/library/socket.html>.
- [16] threading - Thread-based parallelism. (n.d.). Retrieved December 5, 2019, from <https://docs.python.org/3/library/threading.html#threading.Thread.join>.
- [17] time - Time access and conversions. (n.d.). Retrieved December 5, 2019, from <https://docs.python.org/3/library/time.html>.
- [18] datetime - Basic date and time types. (n.d.). Retrieved December 5, 2019, from <https://docs.python.org/2/library/datetime.html>.
- [19] polixo96. (2020, January 13). polixo96/Power-consumption-Datalogger-based-on-python-and-Raspberry-PI. Retrieved from <https://github.com/polixo96/Power-consumption-Datalogger-based-on-python-and-Raspberry-PI>
- [20] __init__. (n.d.). Retrieved December 14, 2019, from https://docs.python.org/3/reference/datamodel.html#object.__init__
- [21] El impacto de la basura electrónica. (n.d.). Retrieved December 27, 2019, from <https://donalo.org/post.php?post=243>
- [22] Product compliance and safety. (2019, October 8). Retrieved January 13, 2020, from <https://www.raspberrypi.org/documentation/hardware/raspberrypi/conformity.md>